the U.S. Patent & Trademark Office
ase stamp the date of receipt of the following
cument(s) and return this card to us.
ty Dkt. No: 00P7500US01 Atty: RSK/cdr
pplicant(s): McDaniel, et al.
:rial No.: 09/809,155
iled: March 5, 2001

X PTO Form 1449 w/2 references
X Information Disclosure Statement
X Date of Deposit: July 2, 2002

To the U.S. Patent & Trademark Office
Please stamp the date of receipt of the following
document(s) and return this card to us.
Atty Dkt. No: 00P7500US01 Atty: RSK/cdr
Applicant(s): McDaniel, et al.
Serial No.: 09/809,155
Filed: March 5, 2001

X PTO Form 1449 w/2 references
X Information Disclosure Statement
X Date of Deposit: July 2, 2002

EXPRESS MAIL

UNITED STATES POSTAL SERVICE®

POST OFFICE TO ADDRESSEE

EV 048644968 US

Customer Copy
Label 11-F August 2000

SEE REVERSE SIDE FOR SERVICE GUARANTEE AND LIMITS ON INSURANCE COVERAGE

EAST PALO ALTO CA
JUL 2 2002

ORIGIN (POSTAL USE ONLY)

PO ZIP Code | Day of Delivery: ☐ Next ☐ Second | Flat Rate Envelope ☐

Date In: Mo. Day Year | ☐ 12 Noon ☐ 3 PM | Postage $

Time In: ☐ AM ☐ PM | Military: ☐ 2nd Day ☐ 3rd Day | Return Receipt Fee

Weight: lbs. ozs. | Int'l Alpha Country Code | COD Fee | Insurance Fee

No Delivery: ☐ Weekend ☐ Holiday | Acceptance Clerk Initials | Total Postage & Fees $

☐ WAIVER OF SIGNATURE (Domestic Only): Additional merchandise insurance is void if waiver of signature is requested. I wish delivery to be made without obtaining signature of addressee or addressee's agent (if delivery employee judges that article can be left in secure location) and I authorize that delivery employee's signature constitutes valid proof of delivery.

NO DELIVERY ☐ Weekend ☐ Holiday
Customer Signature

CUSTOMER USE ONLY

METHOD OF PAYMENT:
Express Mail Corporate Acct. No. A95372 [illegible]

Federal Agency Acct. No. or Postal Service Acct. No.

FROM: (PLEASE PRINT) PHONE ( ) 650 959 9112

SIEMENS CORPORATION
PL [illegible] 7393
1220 CHARLESTON RD
MOUNTAIN VIEW CA 94039-7393

[illegible]D1US01 RSK

TO: (PLEASE PRINT) PHONE ( )

IDS
BOX [illegible]
COMMISSIONER FOR PATENTS AND TRADEMARKS
WASHINGTON DC 20231-0012

FOR PICKUP OR TRACKING CALL 1-800-222-1811 www.usps.com EMS

SIEMENS Corporation
IPD-West Coast
1230 Shorebird Way, Building 4
P.O. Box 7393
Mountain View, CA 94043

PATENT APPLICATION
ATTORNEY DOCKET NO.: 00P7500US01

IN THE
UNITED STATES PATENT AND TRADEMARK OFFICE

Applicants: McDaniel, et al.

Application No.: 09/809,155

Filed: March 5, 2001

Title: PROGRAMMING AUTOMATION BY DEMONSTRATION

CERTIFICATE OF MAILING
Express Mail No.: EV 048644968US
I hereby certify that this paper is being deposited with the United States Postal Service as Express Mail in an envelope addressed to: Assistant Commissioner for Patents, Washington, D.C. 20231, on this date.

7/2/02 Cynthia Reese
Date Signature

THE ASSISTANT COMMISSIONER FOR PATENTS
Washington, D.C. 20231

INFORMATION DISCLOSURE STATEMENT

Sir:

This Information Disclosure Statement is submitted:

[X] under 37 CFR 1.97 (b), or
(Within three months of filing national application; or date of entry of international application; or before mailing date of first office action on the merits; whichever occurs last)

[ ] under 37 CFR 1.97 (c) together with either a:
[ ] Certification under 37 CFR 1.97 (e), or
[ ] A $240.00 fee under 37 CFR 1.17 (p) authorized to be charged to Deposit Account **19-2179** . At any time during the pendency of this application, please charge any fees required or credit any overpayment to Deposit Account **19-2179** pursuant to 37 CFR 1.25.
(After the CFR 1.97 (b) time period, but before final action or notice of allowance, whichever occurs first)

[ ] under 37 CFR 1.97 (d) together with a:
[ ] Certification under 37 CFR 1.97 (e), and
[ ] a petition under 37 CFR 1.97 (d) (2) (ii), required with fee under 37 CFR 1.17(i)(1)
(Filed after final action or notice of allowance, whichever occurs first, but on or before payment of the issue fee)

[ ] under 37 CFR 1.97 (i)
[ ] Filed after payment of issue fee, but before grant of patent - No fee or certification required

Applicant(s) submit herewith Form PTO 1449 - Information Disclosure Citation together with copies of patents, publications or other information of which applicant(s) are aware, which applicant(s) believe(s) may be material to the examination of this application and for which there may be a duty to disclose in accordance with 37 CFR 1.56.

[ ] A concise explanation of the relevance of foreign language patents, foreign language publications and other foreign language information listed on PTO Form 1449, as presently understood by the individual(s) designated in 37 CFR 1.56 (c) most knowledgeable about the content that is given on the attached sheet or by the enclosed English-language search report.

[ ] The undersigned hereby certifies under 37 CFR 1.97(e)(1) that each item of information contained in the information disclosure statement was cited in a communication from a foreign patent office in a counterpart foreign application not more than three months prior to the filing of the information disclosure statement; or

[ ] The undersigned certifies under 37 CFR 1.97(e)(2) that no item of information contained in the information disclosure statement was cited in a communication from a foreign patent office in a counterpart foreign application, and, to the knowledge of the undersigned after making reasonable inquiry, no item of information contained in the information disclosure statement was known to any individual designated in 1.56(c) more than three months prior to the filing of the information disclosure statement.

[X] Applicant does not believe any fee is due. However, at any time during the pendency of this application, please charge any fees required or credit any overpayment to Deposit Account **19-2179** pursuant to 37 CFR 1.25. A duplicate copy of this sheet is enclosed.

The filing of this information disclosure statement shall not be construed as a representation that a search has been made or that no other material information exists. Further, the filing of this information disclosure statement shall not be construed as an admission against interest in any manner or as an admission that the information cited is, or is considered to be material to patentability.

It is requested that the information disclosed herein be made of record in this application.

Respectfully Submitted,

Rosa S. Kim
Attorney for Applicant(s)
Reg. No.: 39,728
Date: 7-2-02
Telephone No.: (650) 694-5330

FORM PTO-1449 (Modified) U.S. DEPARTMENT OF COMMERCE PATENT AND TRADEMARK OFFICE INFORMATION DISCLOSURE STATEMENT BY APPLICANT (Use several sheets if necessary) (37 CFR 1.98(b))	ATTY. DOCKET NO.: 00P7500US01	SERIAL NO.: 09/809,155
	APPLICANT(S): McDaniel, et al.	
	FILING DATE: 3/5/01	GROUP ART UNIT: 2152

U.S. PATENT DOCUMENTS

EXAMINER INITIAL		PATENT NUMBER	ISSUE DATE	PATENTEE	CLASS	SUB-CLASS	FILING DATE if appropriate

FOREIGN PATENTS OR PUBLISHED FOREIGN PATENT APPLICATIONS

EXAMINER INITIAL		DOCUMENT NUMBER	PUBLICATION DATE	COUNTRY OR PATENT OFFICE	CLASS	SUB-CLASS	TRANSLATION	
							YES	NO

OTHER DOCUMENTS (Including Author, Title, Date, Pertinent Pages, Etc.)

/RH/		Smith, David Canfield, Ph.D. thesis "Pygmalion: A Creative Programming Environment" Stanford University, Computer Science, 1975.
/RH/		Wolber, "Pavlov: An Interface Builder for Designing Animated Inferfaces", ACM Transactions on Computer-Human Interaction, Vol. 4, No. 4, pp. 347-386, December 1997.

EXAMINER /Ronald Hartman Jr/ (12/12/2008)	DATE CONSIDERED 12/12/2008

EXAMINER: **Initial citation considered. Draw line through citation if not in conformance and not considered. Include copy of this form with next communication to applicant.**

Pavlov: An Interface Builder for Designing Animated Interfaces

DAVID WOLBER
University of San Francisco

Conventional interface builders provide little support for interactive development of interfaces with application-specific graphics. Some Programming by Demonstration (PBD) systems do provide such support, but none provide full support for demonstrating interfaces, such as those in games, in which the graphics are animated. This article proposes a number of techniques for creating animated interfaces, all of which have been included in an exploratory system, *Pavlov*. Many of the techniques are based on the addition of timing controls to a form of PBD called *stimulus-response demonstration*. Others are based on an adaptation of a traditional animation time-line that integrates end-user interaction with animation. The article also evaluates *Pavlov* with (1) a comparison to other PBD systems in terms of the behaviors that can be specified interactively and (2) a report on an informal user study comparing development in *Pavlov* to development in a conventional interface builder.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*user interfaces*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces; I.2 [**Artificial Intelligence**]: Applications and Expert Systems—*games*

General Terms: Design, Human Factors

Additional Key Words and Phrases: Animation, Programming by Demonstration, user interface design environments (UIDEs)

1. INTRODUCTION

Interactive tools like those in Visual Basic and C++ significantly decrease the time and expertise necessary to build *standard* interfaces, i.e., those with widgets such as menus, buttons, and list boxes. However, these systems provide little interactive support for designing interfaces containing application-specific graphics (arbitrary drawings) and animation.

Programming by Demonstration (PBD) research has shown that many interface behaviors concerning application-specific graphics can be specified without coding [Cypher 1993]. With PBD, instead of writing program code, the designer provides concrete examples that are recorded and

Author's address: Department of Computer Science, University of San Francisco, 2130 Fulton Street, San Francisco, CA 94117-1080; email: wolber@usfca.edu.

generalized by the system. Many PBD systems are based on *programming in the user interface*, which was first introduced in the Smallstar system [Halbert 1984]. With these systems, designers use a drawing editor to draw the graphics, then use the same environment to demonstrate examples of behavior and to execute (test) the program generated by the system. PBD systems include stimulus-response demonstration systems that focus on end-user-triggered behaviors [Fisher et al. 1992; Frank et al. 1995; Gould and Finzer 1984; Myers 1988; Myers et al. 1993; Wolber and Fisher 1991; Wolber 1996], and context-based systems that focus on behaviors that are triggered when objects are in a certain context [Kurlander and Feiner 1993; Repenning 1995; Smith and Cypher 1995].

Though useful for building interfaces such as graphical editors and simulations, current PBD systems do not provide the timing mechanisms necessary to demonstrate timed animation. The most popular interactive methods for defining animation are found in commercial key-frame animation systems such as *Director*. With these systems, designers manipulate a timeline (score) and an interface (stage) to specify the state of each graphic during each time-frame. Interaction and context-triggered behavior can also be specified with these systems, but only by programming with a script language.

The *Pavlov* system was built to explore methods for integrating interface and animation design. Two basic directions were followed: (1) exploring methods for specifying timing during behavior demonstrations and (2) exploring how stimulus-response and context-triggered behavior could be integrated with a key-frame animation timeline. For the former, timing controls and various techniques for demonstrating animation were added to an existing stimulus-response system, DEMO [Wolber and Fisher 1991], and the underlying model and run-time engine of DEMO were redesigned to handle timed behavior. For the latter, a second-level editor was designed that is similar to a traditional animation timeline editor, but has a stimulus-response basis: the (animated) response to each stimulus is viewed in a separate timeline.

The resulting system allows many of the behaviors in games, simulations, and other interactive interfaces to be specified interactively, and with a single uniform framework, i.e., stimulus-response. Designers can not only demonstrate "When I do A, B occurs," they can also demonstrate behaviors such as "When I drag the slider, the car accelerates" and "When I shoot the bullet so that it hits the target, the target and bullet disappear." Some applications, like the driving simulator and arcade game described later in this article, can be developed completely with a combination of demonstration and editing. More complex applications can be prototyped and (when *Pavlov*'s code generation facility is complete) completed by a programmer adding the necessary computational element.

It is this latter purpose which I see as the eventual utility of PBD in commercial interface building. The example applications in this article and others are significant benchmarks in PBD research, but they are not meant as proof that a particular class of applications can be designed without

coding. Most of the interesting, large-scale applications contain a computational element for which current PBD techniques are of no help and for which, one might argue, programming code is the proper medium. A realistic goal is not to eliminate all coding, but to eliminate coding of the visual, interactive part of a graphical animated application. In this way, PBD can provide the same service that Visual C++ provides for applications with standard interfaces.

This article describes the complete *Pavlov* system in detail, for the first time. Section 2 discusses related work. Section 3 introduces *Pavlov* with the sample development session of an arcade game. Sections 4 through 7 provide an overview of the system: Section 4 describes the stimulus-response model, including how context and dynamically created objects are handled; Section 5 describes the timing mechanisms and the techniques for demonstrating animation that were added to the base system; Section 6 discusses the stimulus-response-based timeline editor; and Section 7 provides details of *Pavlov*'s implementation and internal data structures. This overview of the system is followed by a comparison of the interface behaviors that *Pavlov* and other systems cover (Section 8), a report on an informal usability study comparing development in *Pavlov* to that in Visual C++ (Section 9), future directions (Section 10), and a summary.

2. RELATED WORK

An excellent overview of PBD is provided by Cypher [1993]. *Pygmalion* [Smith 1977] and *Tinker* [Lieberman 1981; 1985] illustrated the benefit of "programming" by supplying concrete examples. *Pygmalion* is based on providing examples using special icons for functions, conditionals, and other programming constructs. *Tinker* was the first system to apply PBD to the construction of video games. It is based on the entry of sample *Lisp* function calls. As opposed to *Pygmalion*, it synthesizes conditional statements automatically from the samples and programmer-supplied predicates. Other classic systems include *SmallStar* [Halbert 1984], which introduced the notion of "programming in the user interface" as opposed to a separate environment, and *Metamouse* [Maulsby et al. 1992] and *Eager* [Cypher 1991] which focus on identifying and automating iterative behavior.

Rehearsal World [Gould and Finzer 1984] and *Peridot* [Myers 1988] were early PBD systems that inspired the *stimulus-response* method of PBD. The first systems to allow direct graphical demonstration of a wide range of end-user stimuli and system responses were *DEMO* [Wolber and Fisher 1991] and *Marquise* [Myers 1993]. *DEMO* introduced the stimulus-response model and a technique for demonstrating dynamically created objects, while *Marquise* focused on the demonstration of graphical editors, including those with palettes and modes.

DEMO II [Fisher et al. 1992] and *Inference Bear* [Frank et al. 1995] are stimulus-response systems that allow the designer to perform multiple demonstrations of the same behavior to refine the system's inferences.

Inference Bear uses multiple examples to make sophisticated inferences concerning response parameters—inferred formulas may depend on attributes of arbitrary objects in the interface as well as stimulus parameter values. *DEMO II* uses multiple examples to refine inferences concerning the context for when a response should be executed. It also introduced the use of *guidewires* (annotation objects) that are drawn to aid in demonstrations, but do not appear at run-time.

Pavlov is the first *stimulus-response* system to focus on animation, though there are a few PBD systems not based on stimulus-response that allow some animation to be demonstrated: *Tango* [Lieberman 1981] [Stasko 1991] allows animation paths to be demonstrated in the context of algorithm animation. *Chimera* and *LEMMING* [Olsen et al. 1995] allow interface behavior to be specified with multiple demonstrations of *constraints*, but do not cover time-based animation. Simulation building systems like *KidSim* [Smith and Cypher 1995] and *Agent Sheets* [Repenning 1995] use graphical rewrite rules to allow designers to demonstrate the context for when an operation should be executed. The graphical rewrite rules in these systems allow for context to be specified as virtually any configuration of objects. In both of these systems, the rewrite-rule method of defining context is not integrated with a PBD method of specifying end-user interaction. Thus, games and other *interactive* simulations cannot be designed without coding.

Director is representative of the commercial animation systems that provide facilities for both animation and interaction design. These systems allow animation to be designed quickly and easily using a combination of frame-by-frame animation, in-betweening, and real-time recording. These systems also allow sound and video to be linked into presentations, and they provide a range of features for creating special effects such as *slow-in/slow-out*, *motion blur*, and *squash and stretch*.

Though powerful for defining animation, these systems do not provide a PBD method of defining interaction. The systems allow button-click-triggered animation to be defined in a relatively simple manner. However, more complex stimulus-response behaviors, such as found in an arcade game or driving simulator, require some difficult programming with a scripting language.

A second difficulty in defining interaction with animation systems is that they are based on a single-timeline editor: all the animation sequences of an application are shown on a single timeline. Though such a score is sufficient for noninteractive animation (which was its original purpose), it is too unstructured for applications with interactive as well as time-stimulated animation. Like the programs written before the advent of structured programming (subprocedures), the designer is forced to program control, i.e., where one animation ends and another begins, using *goto* statements. For complex applications with animation sequences triggered by various interactions, the result is a *spaghetti score*.

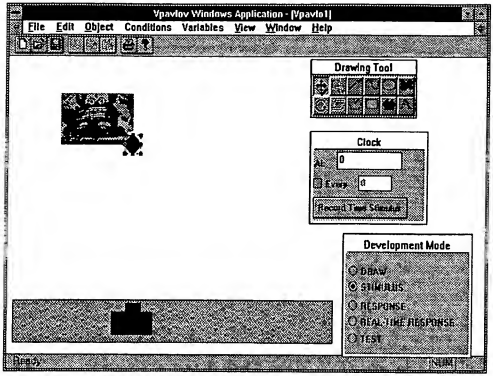

Fig. 1. The Pavlov development of an Arcade game.

3. INTRODUCTORY EXAMPLE

To introduce *Pavlov*, this section provides a sample development session of a typical arcade game. The sample illustrates, by example, how end-user- and context-triggered behaviors can be demonstrated and how timing can be specified to control the speed of animation. The example also illustrates how behaviors can be viewed and edited after they are demonstrated with *Pavlov*'s stimulus-response-based timeline editor.

3.1 Arcade Game Description

In the arcade game, the player (end-user) uses a gun to shoot bullets at monsters that travel across the screen. The player moves the gun using the left- and right-arrow keys; the gun's movement is restricted within an enclosing shooting box. When the player presses the up-arrow key, a bullet is fired from the gun. Monsters are created by pressing the "p" key. When a bullet hits a monster, the bullet and monster disappear. If a bullet does not hit a monster, it disappears when reaching the top of the screen.

3.2 Pavlov Environment

Figure 1 shows the *Pavlov* environment during development of the Arcade game. The basic tools are the menu bar, the drawing editor in the top-right

corner, the clock (middle-right), and the Development Mode Palette (lower right). The designer uses the development modes to inform the system as to whether she is just drawing the interface (Draw mode), demonstrating an end-user or time stimulus (stimulus mode), demonstrating how the system should respond to a stimulus (response or real-time response mode), or testing an interface (test mode). The designer uses the clock to demonstrate when a response operation should be executed relative to the stimulus (using the top *"At"* box) or if an operation should be executed periodically (the middle *"Every"* box). The lower button on the clock, labeled *Record Time Stimulus*, allows the designer to specify that the upcoming demonstrated responses should be triggered by time, not an event.

3.3 Drawing the Initial State

The first task in creating the Arcade game is to draw the gun and its enclosing box. To do so, the designer selects Draw mode from the Development Mode palette and makes use of the graphic primitives and grouping mechanisms available in the drawing mode palette and menu. The designer then names all these objects by selecting each and the menu item Object | Name.

Note that draw mode is only used to draw the initial state of the interface. Thus, the bullets and monsters are not drawn in this mode, since they do not appear until the end-user performs some action.

3.4 Demonstrating Conditional End-User-Triggered Behavior

After drawing the initial state, the designer begins demonstrating the behavior of the interface. First, she demonstrates the right-left movement of the gun. She selects stimulus mode from the development mode palette; then, playing the role of the end-user, she presses the right-arrow key. *Pavlov* reports the recording of the stimulus and then automatically switches to response mode. Before demonstrating the response, the designer selects Conditions | Generate Post-Response, so that *Pavlov* will identify pertinent graphical conditions that must be in effect for the upcoming response to be executed. She then demonstrates the response of moving the gun to the right, say 20 pixels, being sure to keep the gun within the enclosing box. A dialog appears listing the single condition that *Pavlov* identified: "Gun.Within(EnclosingBox)." Only one condition was listed because, at this point, there are only two objects in the interface, and "Within" is the only relationship (of those that *Pavlov* checks) between them. The designer specifies that the condition should modify the stimulus-response, i.e., the move response should only occur when the end-user presses the right-arrow key *and* the gun will be within the enclosing box after (Post-) the move. The following behavior is recorded:[1]

[1]The recorded behaviors listed in this article are described with C++-like pseudocode. The code is used for illustrative purposes and is not actually generated (though we are in the process of generating Java code that has a similar construction).

```
Canvas.OnRightArrowKey
{
   Gun.Move(20,0);   // execute internally
   if (Gun.Within(EnclosingBox))
UpdateView( );
   else
Gun.Move(-20,0);   // undo the move
}
```

The behavior of the left-arrow key is demonstrated in similar fashion. Afterward, the designer enters test mode to run the interface. In test mode, the development tools disappear so that the designer can see exactly what the end-user will see. When test mode is entered, the graphics snap back to their original position, as specified in draw mode. The designer then observes that pressing the left (right) arrow key causes the gun to move left (right) until it reaches the edge of the enclosing box, at which time successive left (right) key presses are ignored.

3.5 Demonstrating Dynamic Allocation and Real-Time Response

Next, the designer demonstrates the bullets being shot. To get out of test mode, the designer presses the "escape" key. She then reenters stimulus mode and presses the up-arrow key. Then, in response mode, she draws a small ellipse (the bullet) above the gun, and names it "Bullet." The designer then switches to real-time response mode and moves the bullet up the gallery at the desired pace, stopping near the top of the screen. The clock runs during the real-time demonstration, so the designer knows that both movement and timing are being recorded.

To demonstrate the final response to the up-arrow-key stimulus, the designer reenters response mode, notes that the clock is still set to its final setting after the real-time demonstration (250 ms), and deletes the bullet. The following behavior is recorded:

```
Canvas.OnUpArrowKey
{
   Bullet bullet = CreateEllipse(205,207,225,229)
   BulletInstances.Add(bullet)
   bullet.Move(0,1) AT 0
   bullet.Move(0,1) AT 2
   . . .. . .
     bullet.Move(0,1) AT 250
   BulletInstances.Remove(bullet)
   bullet.Delete AT 250}
```

When the designer enters test mode, she observes that each time she presses the up-arrow key, a bullet shoots up the screen at the demonstrated

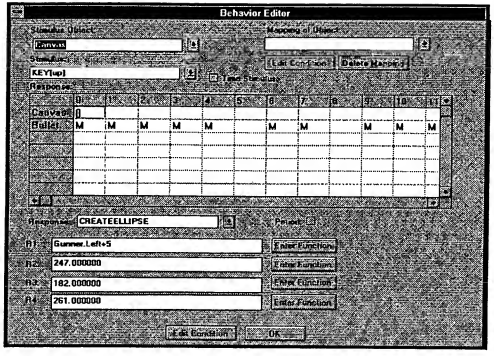

Fig. 2. The Pavlov stimulus-response-based timeline editor.

speed and disappears at the top. Experimenting further, the designer presses the right key to move the gun and then presses the up-arrow key. A bullet shoots up, but it emanates from the original position of the gun instead of its current position.

The designer realizes that *Pavlov* has incorrectly inferred constant values for the parameters of the ellipse (bullet) creation. If *Pavlov* were a system that allowed multiple examples, the designer might proceed to demonstrate more examples of where the bullet should be created. However, *Pavlov* uses a simple, single-example inferencing scheme; to modify the inferred behaviors, the designer uses the second-level editor.

3.6 Editing the Arcade Game Behavior

In this case, the designer uses the editor to modify the parameters so that the bullet will be created relative to the location of the gun. First, she selects the gun and chooses Object | Edit Behavior from the menu. The editor appears (see Figure 2), and the designer selects the stimulus "Key(up)," as shown. The response objects appear in the first column of each row of the timeline. A number of responses appear in the timeline, one for the Canvas.CreateEllipse operation (this is shown as an ellipse), one for each of the series of Bullet.Move operations (shown as "M") that was recorded in the real-time response, and one for the Bullet.Delete operation (not shown in the figure, as it is further to the right in the virtual timeline).

The CreateEllipse operation is selected and appears in the list box labeled "Response" below the timeline. The operation's parameters, shown in the boxes labeled R1–R4, represent the upper-left point (R1 is x; R2 is y) and lower-right point (R3 is x; R4 is y) of the ellipse's "bounding box." They are fixed values, not what the designer intended.

To enter functions for the parameters, the designer first selects the "Enter Function" button next to the R1 parameter, and a dialog appears that allows the designer to select a built-in graphics function as the parameter. The designer specifies that R1, which is the x-value of the upper-left point of the bullet, should be computed as "Gun.Left+5," i.e., five pixels to the right of the left edge of the gun. The same method is then used to change R2 (upper-left.y) to read "Gun.Top," R3 (bottom-right.x) to read "Gun.Right-5," and R4 (bottom-right.y) to read "Gun.Top+10."

The designer closes the editor and enters test mode. The up-arrow key now causes bullets to emanate from directly above the gun, even after the gun is moved.

Next, the designer specifies how monsters are created. First, she demonstrates a stimulus of pressing the "P" key. Then, in response mode, she creates a rectangle of the desired size, then selects Object | Set Bitmap from the menu, and enters the file name of an appropriately foul monster bitmap. Then, while still in response mode, she selects the "Every" check box on the clock, sets the period to "1," and moves the monster a small amount to the right. The following behavior is recorded:

```
Canvas.OnPKey
{
   Monster monster =CreateRect(0,10,20,30)
   MonsterInstances.Add(monster)
   monster.SetBitmap("monster.bmp")
   monster.Move(0,4) Every 1
}
```

In test mode, the designer can create bullets and monsters with the up arrow and "P" key, respectively. The bullets disappear upon reaching the top of the screen, since a deletion was demonstrated, but the monsters never die: when they reach the right side of the screen they reappear on the left and continue to move (by default objects wrap around the screen instead of continuing on in the nonvisible virtual world).

3.7 Demonstrating Context-Triggered Behavior

The final behavior is that of bullets killing monsters. First, while still in test mode, the designer creates a monster and a bullet in the just-demonstrated manner, then selects the "escape" key before the Bullet instance is deleted. Next, the designer selects Conditions | Generate Pre-Response from the menu. She then demonstrates a stimulus of moving the bullet, making sure to have it intersect the monster, as in Figure 1. *Pavlov*

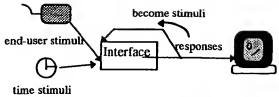

Fig. 3. Stimulus-response machine.

lists the conditions it finds, in this case just the Bullet.Intersects(Monster) condition. The designer specifies that the condition should modify the behavior. Finally, in response mode, she deletes the bullet and the monster, and the following behavior is recorded:

```
Bullet.OnMove(x,y))
{
   for each monster in MonsterInstances
     if (this.Intersects(monster)
         {
       BulletInstances.Remove(this)
       this.Delete
       MonsterInstances.Remove(monster)
       monster.Delete
     }
   }
```

In test mode, the designer observes that she has created an arcade game. She can create any number of monsters and shoot them with the gun. Each time a monster is shot, it and the bullet disappear.

4. THE STIMULUS-RESPONSE MODEL

The Arcade game example introduces many features of the stimulus-response model, including interaction, context-driven behavior, and animation. This section describes the model in more general terms in order to bridge the gap between the specific and the general, i.e., persuade the reader that *Pavlov* is useful for designing many animated interfaces, not just arcade games.

In the internal *Pavlov* model, an interface is viewed as a stimulus-response machine (Figure 3). Stimuli are either physical actions (e.g., drag the mouse with left button down), higher-level operations (e.g., Move, Rotate), or time. The interface responds to stimuli by executing a set of timestamped operations. Operations either create, transform, or delete objects. The set of operations includes the primitives found in most drawing editors (e.g., Move, Rotate, Resize, SetColor, etc.) and two additional primitives: *MoveForward* and *SetImage*. *MoveForward* allows an object to be moved while constrained to the vector defined by its direction attribute

(see Section 5.2). *SetImage* associates an object with a bitmap image and thus allows for key-frame animation in which an object continually changes form (e.g., a person running). The basic drawing editor operations and the two additional primitives offer the *base* functionality necessary to demonstrate many interface behaviors. Because of the generality of the stimulus-response model, other primitives, such as sound or video, could also be added.

After a response operation is executed, it becomes a stimulus that can itself cause another response (as shown above). Thus, a chaining of events can occur. For instance, in the Arcade game, the stimulus of pressing the up arrow key causes a response of the bullet being created and moved periodically up the gallery. After the execution of each move of the bullet, the move operation becomes a stimulus, i.e., *Pavlov* checks if there is a response that should occur as a result of the move. In this case, there is a conditional response: if the move leads the bullet to intersect the monster, it causes a response of deleting the monster and bullet.

4.1 Stimulus-Response Demonstration

The challenge of a stimulus-response development system is to provide clear syntax and semantics for how the designer uses the set of physical actions and operations to demonstrate the behavior of the target interface. The syntax of *Pavlov* is straightforward: the designer changes development modes to inform the system whether the intent of the following actions is to draw, demonstrate an end-user stimulus, demonstrate a system response, or test the "target" interface. This syntax was first used in DEMO and has been retained in the stimulus-response systems DEMO II and *Marquise*. *Inference Bear* uses a slightly different syntax: the developer supplies a "Before" snapshot of the interface, an "After" snapshot of the interface, and an event that causes the transition from the Before to the After picture. This syntax appears to be essentially the same as *Pavlov*'s modal system, though no usability comparisons have been performed.

Providing clear semantics is a more challenging problem: the goal is for the system to always record a stimulus-response descriptor that perfectly matches the intent of the designer's demonstration (this is the primary challenge of all PBD systems). *Pavlov* uses an *explanation-based learning* approach: from a single demonstration of a stimulus-response pair, the system uses domain knowledge and the information provided by the demonstration to record as reasonable (to the system) a stimulus-response descriptor as possible. The approach is based on the following ideas:

—*A Simple Inferencing Scheme*: The inference scheme is kept as simple as possible so that even if the inferences made are not intuitive to a particular designer, the designer can quickly learn and understand them.

—*Single Example*: One demonstration is sufficient for some behaviors which require multiple examples in other systems. The reason is that *Pavlov* uses domain knowledge to make educated guesses as to a designer's intent, whereas other systems do not [Frank et al. 1995; Olsen et al.

1995]. If domain knowledge is not used, even simple behaviors, such as mapping a mouse drag to the movement of an object, require more than one demonstration [Frank et al. 1995]. The goal of *Pavlov* is to ease the task of demonstrating common behaviors, at the price of not allowing less common behaviors to be completely specified by demonstration.

—*Reliance on the Second-Level Editor*: The most important utility of the system is to allow stimulus-response behaviors to be generated automatically, so that programming concepts such as event loops, event dispatching, idle events, and timer interrupts are hidden from the designer. The details of these behaviors (e.g., response parameters) can sometimes be inferred with a single demonstration, and more often with multiple demonstrations, but a computational element is often necessary. Thus, second-level editing of inferred behaviors is considered part of the normal design process.

4.1.1 *The Semantics of a Stimulus Demonstration.* When a stimulus is demonstrated, the designer specifies two things: that the end-user, at run-time, can perform that same operation (e.g., the end-user can move a certain object), and secondly, that the upcoming demonstrated response (which is optional) will be executed, at run-time, in response to the stimulus. At run-time, the end-user will only be able to use the operations that the designer explicitly demonstrated as stimuli and will only be able to access those operations using the physical action (e.g., mouse up, key press) that was used in the demonstration. For example, in the Arcade game, the end-user can only invoke activity by pressing the "P" key or the arrow keys, because only those keys were demonstrated as stimuli. The designer cannot manipulate the gun with the mouse, because no such stimulus was demonstrated. This positive-example method of specifying the functionality of the system is in contrast to the scheme of Olsen et al. [1995] in which the designer "freezes" the objects that cannot be manipulated.

To demonstrate a key-press stimulus, the designer simply enters stimulus mode and hits the appropriate key. To demonstrate a mouse click, she chooses selection mode from the drawing editor and clicks on a particular object or on the canvas. After these demonstrations, the mode is set to response mode, and the *current stimulus* is set so that successive demonstrated responses are associated with it.

Mouse-drag-based stimuli demonstrations are interpreted in a slightly different manner than simple mouse clicks or key presses. When a designer selects an operation such as Move from the drawing palette and drags an object, *Pavlov* assumes she has two intentions: (1) to show that the end-user can invoke the operation at run-time by dragging the mouse on the object with the same mouse button and auxiliary keys pressed and (2) to define the operation as the *current stimulus* so that, in response mode, she can show how the other objects should react to the stimulus object being moved.

Internally, *Pavlov* records an implicit stimulus-response that maps the physical operation (e.g., drag object1 with left mouse down) to the higher-

level operation (e.g., move) and sets the higher-level operation as the *current stimulus*. Often, the designer is not even aware of the implicit recording: she is concerned with demonstrating a reaction to the higher-level operation. At other times the designer is aware and does not even plan to demonstrate any responses in response mode, e.g., she just wants to show that, at run-time, the end-user can rotate a certain object by dragging it with the left mouse button pressed (and with no reaction from other objects).

This approach of implicitly inferring the physical stimulus-response mapping *and* recording the explicitly demonstrated higher-level stimulus eases the task compared to other systems which require separate demonstrations, e.g., one demonstration to show that dragging the mouse on an object moves it and one to show that moving the object causes some other object to rotate [Piyawadee and Foley 1995]. A drawback of this implicit approach is that the designer must sometimes remove mappings that are not desired. For instance, in the Arcade game, the designer demonstrates a stimulus of moving the bullet over a monster and a response of "killing" the monster. From the demonstration, *Pavlov* not only infers the intended behavior, but also infers that the end-user can drag the mouse on the bullet to move it. Since the end-user should not be able to drag a bullet, the designer must use the editor to delete the mapping.

4.1.2 *The Semantics of a Response Demonstration.* When the designer demonstrates an operation R on object O2 in response mode, the system connects R to the previously demonstrated stimulus S:

```
O1.OnS(s1,s2,. . .)
{
    O2.R(r1,r2,. . .)
}
```

Even if the designer's demonstrations are not generalized, the ability to demonstrate stimulus-response mappings eliminates the programming necessary to produce the framework for a graphical program. *Pavlov* does generalize from demonstrations, however, using an *explanation-based* approach: from a single demonstration of a stimulus-response pair, the system uses domain knowledge and the information provided by the demonstration to record as reasonable (to the system) a stimulus-response descriptor as possible. The theory behind the approach is that it is most important to allow stimulus-response mappings to be recorded in a straightforward manner: if the inferred mapping is as desired, fine: if not, the designer can use the editor to *adjust* the mapping.

Much of the inferencing in a stimulus-response system involves determining the formula for the run-time response parameters r_i. The simplest solution is, of course, to execute a response during execution with the same parameters as were demonstrated, e.g., move some object 40 pixels if it was moved 40 pixels during the response demonstration. This is the solution *Pavlov* uses when the corresponding stimulus has no parameters (e.g., the stimulus is the pressing of some key).

However, for stimuli that do have parameters, it is often the case that the reaction is proportional, i.e., the response parameters r_i are proportional to the stimulus parameters s_i. For instance, in a driving simulator, a car is rotated an amount proportional to the amount the steering wheel is rotated, or in a Celsius-Fahrenheit converter, the Celsius gauge is modified an amount proportional to the amount the Fahrenheit gauge is modified. When a stimulus and response transformation operation are demonstrated, and both have parameters, *Pavlov* infers proportional constants $C_i = r_i/s_i$ that relate the parameters of the stimulus to those of the response. The behavior is recorded as

```
O1.OnS(s1,s2,. . .)
{
    O2.R(C*1s1, C2* s2. . .)
}
```

The formula for R illustrates that the system infers the first parameter of the stimulus to be related to the first parameter of the response, the second to the second, and so on. The basis of this inference is that most graphics operations either have a single parameter or have two parameters denoting x and y coordinates, so in practice the corresponding stimulus and response are often related.

When objects are created as a response, the proportion scheme is not used. For creation responses, absolute values are used for the drawing parameters unless the stimulus is a button click. If the stimulus is a button click, the (x,y) coordinates of the click are used as offsets to the drawing parameters. For example, if the designer demonstrated a stimulus of clicking at location (200,200) and then created a circle with radius 100 centered at (200,200), the system infers the behavior:

```
Canvas.OnButtonUp(x,y)
{
   Ellipse.Create(x−100, y−100, x+100, y+100)
}
```

Further, at run-time, the ellipse is created relative to the location of the click. Note that this scheme only infers behaviors in which the creation is relative to the mouse click; it does not handle behaviors like those in the Arcade game, in which an object (the bullet) is created relative to the location of some other object (the gun).

Pavlov's simplistic inferencing strategy differs from other systems that allow a designer to refine behaviors through multiple demonstrations [Fisher et al. 1992; Frank et al. 1995; Kurlander and Feiner 1993; Myers et al. 1993; Olsen et al. 1995]. Such *empirical-based* learning approaches allow more complex behaviors to be demonstrated. For example, *Inference Bear* [Frank et al. 1995] can infer response parameter formulas that are functions not only of stimulus parameters but of attributes of arbitrary

objects in the interface, e.g., it can infer that when a button is clicked, an object moves an amount equal to the width of another object:

```
Object1.OnLeftMouseUp(x,y)
{
  Object2.Move(Object3.Width( ),0)
}
```

A drawback of a multiple-example approach is that it complicates both the demonstrational interface and the semantics of the demonstrational language. In usability testing of *Inference Bear* [Piyawadee and Foley 1995], the demonstration of behaviors which required multiple examples was noted as significantly more difficult than those requiring only a single example. This finding is supported by this author's own experience with multiple example versions of DEMO.

Part of the difficulty may be that a single stimulus-response demonstration environment is at or near the limit of complexity that most users can handle. In observing novice *Pavlov* users, it has been found that the demonstrational interface is challenging: the drawing editor interface that users are quite comfortable with is augmented by another mode, the development mode (Draw, Stimulus, Response, Test). And though most adapt quite easily after a few hours, adding another mode that specifies whether a second example should refine a behavior (as in multiple example systems), overwrite it, or extend it (i.e., add additional responses to a stimulus) is problematic. Furthermore, the inferencing rules necessarily become more complicated for a designer to learn.

Some research questions arise. Does the set of behaviors that can be defined only with multiple examples warrant the added complexity? And to what degree does the existence of a second-level editor (such as the one discussed in Section 6) lessen the need for a multiple example system?

4.2 Demonstrating Context

Some actions are not executed when a particular end-user stimulus occurs, but when objects of the interface are in a particular state. Such context-driven behaviors are the basis for PBD graphical rewrite systems [Repenning 1995; Smith and Cypher 1995] and constraint-based systems [Kurlander and Feiner 1993]. Though these systems are powerful for defining context-triggered activity, they are not integrated with mechanisms for defining end-user triggered activity.

Pavlov provides a mechanism for demonstrating conditional behaviors that is fully integrated within the stimulus-response framework. A condition C is defined only within the context of a stimulus-response behavior:

```
O1.OnS(s1,s2,. . .)
{
  if (C) O2.R(r1,r2,. . .)
}
```

For example, in the Arcade example, the developer demonstrates a stimulus of moving the bullet *so that it intersects the monster*, then demonstrates the response of deleting the bullet and the monster.

This model for handling context was not apparent when the stimulus-response framework was first designed. Originally, a model similar to that used in constraint-based systems was envisioned: the designer would select a special "condition stimulus" mode and the relevant objects, causing the system to identify and list textual descriptions of the relationships between the objects, then choose one or more to define the condition. The designer would then demonstrate the response that should fire when the condition became true. A behavior of the form "On C→O.R" would be recorded.

However, such a constraint-like scheme would complicate the demonstrational interface. There would have to be a special mode or mechanism for demonstrating "condition stimuli," thereby reducing the simplicity and uniformity of the stimulus-response model. Furthermore, the scheme is not particularly efficient because, at run-time, the system must check a potentially large set of conditions after the execution of *each* operation, to see if the operation caused the condition to become true.

Instead of a general constraint-checking scheme, a condition in *Pavlov* is defined as a modifier of a stimulus-response. The model is kept more coherent, and the specific condition need only be checked after the execution of the specific stimulus operation. Such an efficient method is especially important in an animated interface, as conditions must be processed on-the-fly.

One argument against the approach used in *Pavlov* is that a designer must redemonstrate the condition for each type of stimulus that can cause it to occur. In practice, however, many behaviors require only one demonstration (e.g., the only stimulus that causes a bullet to hit a monster is a move operation).

A conditional stimulus-response is demonstrated by first toggling on the *condition checker*, then demonstrating a stimulus operation and a response operation that leave the interface in a state in which the condition is true.

When the condition checker is on, *Pavlov* tests the following graphical conditions relating two objects:

—Intersects
—Within
—Encloses
—Endpoint1Within
—EnclosesEndpoint1
—Endpoint2Within
—EnclosesEndpoint2

It then lists those that it finds to be true in the *condition editor*. The designer can choose a single condition or compose a complex relation by using AND and OR operators. She may also negate a condition, i.e., specify that a response be executed only if a condition is false.

The set of conditions checked in *Pavlov* is not definitive: the set will be refined as more interfaces are created by users. The research goal is to determine a set that provides enough expressive power without requiring the designer to choose from a huge set of system-identified conditions.

One method used by *Pavlov* to limit the system-identified conditions is to test only relationships concerning the stimulus object. In practice, most conditions concern the stimulus object, since the condition becomes true after its execution. For conditions that do not involve the stimulus object, the designer must use the editor to enter the desired condition.

4.2.1 *Pre- and Postresponse Conditions.* Conditional behaviors are complicated by the fact that for some, the condition should be evaluated immediately after the stimulus operation (*Preresponse* condition), and for others, it should be evaluated after the response operation (*Postresponse* condition). In *Pavlov*, the designer chooses one of these types when turning on the condition checker.

The Arcade game behavior

```
Bullet.OnMove(x,y))
{
for each monster in MonsterInstances
   if (this.Intersects(monster)
   {
      BulletInstances.Remove(this)
      this.Delete
      MonsterInstances.Remove(monster)
      monster.Delete
   }
}
```

provides an example of a preresponse condition. The condition "Bullet.Intersects(Monster)" is checked immediately after each occurrence of the move stimulus, and the responses are only executed if the condition is evaluated to true.

The left-right behavior of the gun provides an example of a postresponse condition:

```
Canvas.OnRightArrowKey
{  Gun.Move(20,0)  // execute internally
   if (Gun.Within(EnclosingBox))
   Update( );
   else
   Gun.Move(120,0) // undo the move
}
```

When one of the arrow keys is pressed, the system internally executes the response of moving the gun, without updating the display, then checks if the resulting gun is within the enclosing box. If the condition is true, the display is updated; if the condition evaluates to false, the operation is undone, and no change is made to the display. Note that if a preresponse condition had been specified for this behavior, a right-arrow key pressed with the gun just inside the edge of the box would cause the gun to be moved outside the box.

A third possibility exists for the evaluation of conditions, though it is not yet implemented. In some behaviors, a stimulus should cause a response only if a condition is false before the stimulus, but becomes true after the stimulus is executed (we call this type of condition *justBecameTrue*). For instance, consider an arcade game in which a bullet moving over some object causes a point to be added to a score, but does not delete the object or the bullet. On initial impact, when the bullet first intersects the monster, the score should be incremented. However, since the animated movement of an object is generally fostered by the execution of numerous small move operations, some number of successive moves will find the bullet still intersecting the monster. Clearly, for this example and similar ones, only the initial move operation that causes the intersect condition to become true should cause the demonstrated response.

Usability tests have shown that concepts such as pre- and postresponse conditions are not intuitive to most first-time *Pavlov* designers; they are complicated and require training. However, once this training has been given, most designers have understood the concepts.

4.2.2 *Variable Conditions.* Besides graphical conditions, the designer can also specify program variable conditions by demonstration. Integer variables can be defined in *Pavlov* by selecting Variables | Create New in the menu. Prior to a demonstration, the designer sets a variable to a certain value. When the condition checker lists the graphical conditions it finds, it also lists variable conditions. For instance, if a variable "Mode" is set to 2, *Pavlov* lists the condition "Mode=2" in the condition editor. The designer can choose the condition or specify a different relational operator for it.

The form-based use of simple variables (or modes) is common to many PBD systems, including *Marquise* [Myers et al. 1993] and *KidSim* [Smith and Cypher 1995]. *Marquise* also provides a special mechanism for creating palettes that the end-user uses to control the value of variables.

4.2.3 *Demonstrating Context in Other Systems.* Context is demonstrated in a similar manner in DEMO II [Fisher et al. 1992], though multiple examples can be provided. Context is based on modifying stimulus-response code by hand in *Inference Bear* [Frank et al. 1995]. The *Marquise* system provides the ability to demonstrate mouse location as a context for when a response is executed. As a mouse drag stimulus is demonstrated, the system draws special icons representing mouse positions. The designer can then demonstrate responses that should occur when the mouse is dragged to that position.

It is interesting to compare context demonstration in *Pavlov* to that of a graphical rewrite system like *KidSim*. *KidSim* does not identify any of the conditions that *Pavlov* does: it identifies context based on a two-dimensional grid, i.e., a condition is true when the grid locations surrounding an object are configured in a particular way. *Pavlov* objects are not restricted to grid cells, and conditions are not configurations of cells, but instead concern one or more relationships concerning two objects.

Many behaviors can be inferred by either type of system. A *KidSim* designer could demonstrate that a fish is eaten if one of the cells around it contains a shark, though the shark and fish must both fit within a single cell. A *Pavlov* designer could specify basically the same thing, by demonstrating that the fish is eaten when a shark *intersects* the fish from any angle. The *Pavlov* method provides more flexibility in that there is no restriction on the size of the shark or fish.

However, *KidSim* can identify conditions based on more complex grid configurations, such as that the fish is eaten only if it has a shark to the top-left cell of it. *Pavlov* is not based on a grid, so there is no way to demonstrate such relationships as "shark in the top-left cell."

Because *Pavlov* identifies object-object relationships, it can identify some conditions that *KidSim* cannot. For instance, a *Pavlov* designer could demonstrate that a fish can move only within a pool of circular or other shape. In *KidSim*, the pool would have to be rectangular in nature.

More research is necessary to distinguish, in detail, (1) the expressive power of each scheme and (2) if it is possible for the two approaches to be integrated in some manner, so that as many conditions as possible can be demonstrated.

4.3 Dynamically Allocated Objects

Dynamically allocated objects are those objects that do not appear at the beginning of execution, but are instantiated after execution begins. A system must allow the designer to demonstrate both (1) how the objects are created (and deleted) and (2) the behavior of such objects after they are created. The second concern is of particular interest, because the designer must demonstrate with example instances, and the system must generalize the behavior to the whole "class" of instance that might be created at run-time.

4.3.1 *Demonstrating Object Creation.* The creation of an object is demonstrated in two ways with *Pavlov*: the creation is demonstrated as a response to some stimulus, in which case each occurrence of the stimulus at run-time causes an instance of the same size and shape to be created, or the designer demonstrates the creation as a stimulus, in which case the end-user, at run-time, can drag the mouse to create and size a new instance of the object (this object must be a primitive shape such as a rectangle or line).

The Arcade game provides an example of the first method: the designer demonstrated a press of the up arrow key as a stimulus and created a bullet of a certain size as the response. At run-time, a new instance of the

bullet is created each time the end-user presses the arrow key. An example of the second method can be found in the development of a diagram editor. The designer demonstrates a stimulus of creating a rectangle, perhaps with no response. At run-time, when the end-user drags the mouse on the canvas, she is allowed to create and size a new rectangle instance. If desired, variables and variable conditions can be used to allow different shapes to be drawn when the mouse is dragged, depending on the value of some variable representing the "drawing" mode.

4.3.2 *Demonstrating the Behavior of Dynamically Created Objects.* Though the semantics for demonstrating how dynamic objects appear is straightforward, the semantics for demonstrating their behavior *after they appear* is more complex. During development, after a dynamically allocated object is created, the designer may demonstrate stimulus-response behaviors on that object. But there is not a one-to-one mapping between the development-time object and a run-time object: instead, the development-time object represents the class of all instances that might be created at run-time in the same manner that the representative was created. Thus, the system must infer what the designer intends when she demonstrates a stimulus or response on a representative object.

When a stimulus operation is demonstrated on a representative object, the inference is straightforward: at run-time, the same stimulus on any instance of the class should fire the response. However, when a response is demonstrated on a representative object, a more complex inference scheme is necessary. The most simplistic inference is that, at run-time, the response should be executed on all instances of the class. Such an inference is correct if, for instance, the designer clicked on a button labeled "Color Circles Red" as a stimulus, then demonstrated a color change for a representative circle.

However, there are many behaviors in which the response should only be executed on selected instances. *Pavlov* infers selection criteria in a couple of ways. First, the condition mechanism can be used to select the instances that a response should be executed on. For instance, in developing a diagram editor, the designer might demonstrate (with condition checker on) a stimulus of moving a rectangle and a response of reshaping a connecting line so that it stays connected to the rectangle. The line, of course, is a representative object. When the system identifies the condition of "Rect. EnclosesEndpoint1 (Line)," the designer selects it, and the following behavior is recorded:

```
Rect.OnMove(s1,s2)
{
   For each line in LineInstances
      if (this.EnclosesPoint1(line))
         line.ReshapePoint1(1*s1,1*s2)
}
```

At run-time, when a rectangle is moved, the system will loop through the list of line instances and only execute the reshape response on the instances of Line that indeed intersect the rectangle of the stimulus (the reader is referred to Wolber and Fisher [1991] for details of this example).

The system also makes some inferences for specific types of demonstrations on representative objects. If a stimulus and response are demonstrated on the same object, which is an instance of a class, a flag is set so that at run-time the demonstrated response is only executed on the instance which is the stimulus object.

For instance, in the Arcade game example, the designer demonstrated a stimulus of moving a particular bullet, then a response of deleting that same bullet instance. *Pavlov* recorded the following behavior:

```
Bullet.OnMove(x,y))
{  for each monster in MonsterInstances
     if (this.Intersects(monster)
     {
       BulletInstances.Remove(this)
       this.Delete
       MonsterInstances.Remove(monster)
       monster.Delete
     }
}
```

so that at run-time, when the move stimulus occurs and the condition Bullet.Intersects(Monster) evaluates to true, the response of deleting the bullet is only executed on the particular bullet that caused the stimulus, not all existing instances of the bullet class. Note that if the designer would have demonstrated the deletion of a bullet instance different from the stimulus object, *Pavlov* would have generated code so that all bullets would be deleted.

```
Bullet.OnMove(x,y))
{
   for each monster in MonsterInstances
     if (this.Intersects(monster)
     {
       BulletInstances.RemoveAll
       . . .
     }
}
```

Another inference *Pavlov* makes is that if the demonstrated response object is the same instance as one of the objects in a conditional, at run-time the response is executed only on the instance that is part of the evaluated condition. Thus, in the behavior above, only the instance of Monster that the bullet hits is deleted.

A third inference made by the system is that if a response operation is demonstrated on an instance that was created earlier in a response sequence, at run-time the operation is only executed on the last created instance. For example, the demonstrated response to the up-arrow key was the creation of a bullet followed by numerous moves of the bullet. When the interface is executed, *Pavlov* only applies the move to the newly created bullet, not all existing bullets.

4.3.3 *Dynamically Created Objects in Other Systems.* This mechanism for demonstrating the behavior of dynamically created objects was introduced in the author's *DEMO* system, and similar approaches are used in other stimulus-response systems [Fisher et al. 1992; Frank et al. 1995; Myers et al. 1993]. *KidSim* allows object creation to be demonstrated through graphical rewrite rules, and it allows rules to be applied to any instance of a "class" of objects.

5. DEMONSTRATING TIMING AND ANIMATION

Systems for demonstrating animation have existed for over 25 years [Baecker 1969]. *Pavlov*'s contribution is the integration of animation demonstration with the stimulus-response model for defining interaction. This integration allows for the demonstration of animation triggered by various stimuli, and it allows for the demonstration of animation in which the direction and acceleration of an object are controlled by the end-user. To illustrate, we present a version of the driving simulator example first introduced by Wolber [1996] and shown in Figure 4.

5.1 Example of a Driving Simulator

In the driving simulator, the top car begins moving when the program begins, follows a predefined path around the track, then stops near its starting point. The bottom car begins moving only when the driver (end-user) rotates an "accelerator." Its speed and direction are controlled by the driver manipulating the accelerator and the steering wheel.

5.1.1 *Specifying the Direction Attribute of the Car.* The first task in creating the driving simulator is to draw the cars, the road, the accelerator, and the steering wheel. Next, the designer begins specifying the behavior of the interface. Because she wants the cars to move as most vehicles do, she selects each, chooses Object | Set Direction from the menu, and enters an angle that defines the *direction attribute* of the particular car. Since the cars initially faces straight ahead on the x-axis, the designer sets the angle to 0. A vector emanating from its center appears on each car to signify that it will only move forward and backward in relation to its direction, and must be rotated to change direction. The vector does not appear during execution of the target interface.

5.1.2 *Demonstrating a Time-Stimulated Behavior.* The designer is now ready to demonstrate the stimulus-response behavior of the top car. In this case, the stimulus is time: at time 0 (the beginning of execution) the car

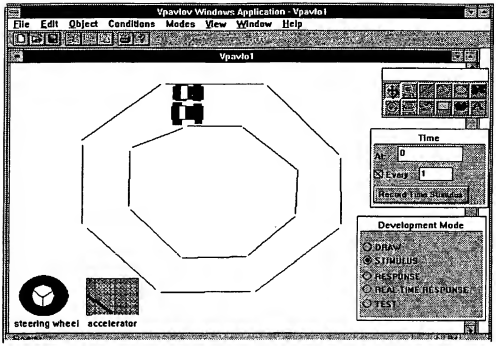

Fig. 4. Pavlov development of a driving simulator.

should begin moving. To demonstrate, the designer selects stimulus mode, sets the clock *At:* box to 0, and clicks on the Record Time Stimulus button. The system reports that a *time stimulus* has been recorded, and automatically switches to response mode. For this behavior, the designer wants to demonstrate a real-time response, so that mode is selected. The designer then selects Move in the drawing palette and drags the car around the track. As she drags the mouse, the car does not ever move diagonally, but instead moves its nose toward the mouse caret and rotates its base in order to turn. After the designer releases the mouse button, she observes (in the editor) that the system has recorded a series of discrete, timestamped responses, made up of alternating MoveForward and Rotate commands.

Next, the designer enters test mode to visually test the demonstrated behavior. Immediately, the top car begins moving along the same path that was demonstrated.

5.1.3 *Demonstrating Acceleration.* The designer then turns her attention to the bottom car. The bottom car's behavior is not triggered by time, but by an end-user action. Thus, instead of demonstrating a time stimulus, the designer plays the role of the end-user and demonstrates an action. After entering stimulus mode, she selects the Rotate icon in the drawing palette, presses the left mouse button on the rectangle denoting the accelerator, and rotates it clockwise some amount, say, negative 0.36 radians. The system reports that a stimulus was recorded and automati-

cally switches the development mode to Response. The system also records an implicit stimulus-response descriptor mapping the physical mouse action to the higher-level operation:

```
Accelerator.OnLeftMouseDrag(int x,int y)
{
   radians = Canvas.CalcRadiansFromDrag( )
   Accelerator.Rotate(radians)
}
```

Next, the designer needs to demonstrate that the rotation of the accelerator should cause a response of accelerating the movement of the bottom car. First, she inputs "1" in the *Every:* box of the clock. She knows that this will cause the upcoming demonstrated response to be executed periodically every time-frame in the target interface. Next, she moves the car some amount, say, 17 units. Because the car is a "directed" object, the car's movement is restricted: it can only be moved on the vector defined by its direction arrow. This restriction is as the designer desired: in response to the rotation of the accelerator, she wants the car to *move forward*, not change direction. The system reports the recorded stimulus-response descriptor containing the inferred proportional constant ($-47.22 = 17/-0.36$).

```
Accelerator.OnRotate(s1)
{
  BottomCar.MoveForward(−47.22*s1) Every 1
}
```

Next, the designer enters test mode to check the interface. The top car immediately begins moving in the demonstrated path. The designer, playing the role of the end-user, rotates the accelerator. The bottom car begins moving and continues to move even after the designer releases the mouse from the accelerator. As the car leaves the right side of the screen, it reappears on the left. The designer again experiments with rotating the accelerator and notices that rotating it clockwise speeds up the car, while rotating it counterclockwise slows it down.

5.1.4 *Demonstrating a Change in Direction.* The next task is to specify the behavior of the steering wheel so the end-user can control the direction of the bottom car. The designer enters stimulus mode and rotates the steering wheel. Then, in response mode, she sets the *Every:* box in the clock to 1 and rotates the bottom car. The following descriptor is recorded:

```
Wheel.OnRotate(s1)
{
    BottomCar.Rotate(0.25*s1) Every 1
}
```

To test this new behavior, the designer once again enters test mode. The top car immediately begins its path. The designer rotates the accelerator to get the bottom car moving, then releases the accelerator and rotates the steering wheel to control the car's direction. She observes that, just like a real one, the car continues to turn if it is held steady at a position other than its original setting (this is due to the periodic rotation of the bottom car).

5.1.5 *Demonstrating a Restricted Acceleration.* The designer continues to test the interface and soon realizes that if she rotates the accelerator counterclockwise past its origin (and outside the enclosing rectangle), the car begins moving backward. To alleviate this problem, she uses the editor to delete the previously recorded accelerator behavior and then redemonstrates it. First, in draw mode, she sets the top-left point of the accelerator on the left edge of the enclosing rectangle; this defines its initial setting. Next, she chooses Conditions | Generate Pre-Response from the menu and demonstrates the stimulus of rotating the accelerator. A dialog appears, listing a set of graphical conditions relating the stimulus object to other objects in the interface. The designer selects the condition "Accelerator. Within (EnclosingRectangle)," and the system records a modified version of the originally recorded stimulus-response descriptor:

```
Accelerator.OnLeftMouseDrag(int x,int y)
{
   if (this.Within(EnclosingRectangle))
   {
      radians = Canvas.CalcRadiansFromDrag( )
      Accelerator.Rotate(radians)
   }
}
```

The designer proceeds to demonstrate the response of moving the car forward, as she did in the first iteration. Afterward, she reenters test mode and observes that when the accelerator is rotated clockwise, then counterclockwise back to its origin on the left-edge of the enclosing rectangle, it cannot be rotated any further. Because in draw mode the initial state of the accelerator was demonstrated as this left-edge, the car stops; there is no way to make the car move backward.

5.2 The Stimulus-Response Model of Animation Demonstration

The driving simulator illustrates how animation is integrated into the stimulus-response model. In general, an animation path is demonstrated with a real-time response demonstration, a series of timestamped response demonstrations (the editor can be used for in-betweening), or with a periodic response demonstration. Like any other response, an animation path can be triggered by any kind of end-user or time stimulus.

Real-time response demonstrations, such as was used to show the movement of the bullet in the Arcade example and the movement of the top car in the simulator, allow the designer to directly demonstrate the velocity of a path. As soon as the designer begins dragging an object, the clock is started, and *Pavlov* records a series of timestamped transformations. At run-time the series is replayed. Because operations are recorded instead of picture frames (as in traditional animation systems), the recorded path is not constrained to a particular starting point, object, or background. Thus the designer can use the editor to reuse paths. The mechanism is also slightly more general than in systems such as Director, because any operation, not just a move, can be demonstrated in real-time.

Pavlov also allows the equivalent of key-frame animation through the demonstration of a series of timestamped responses. The designer just increments the clock prior to each demonstration, and *Pavlov* places a timestamp on the recorded operation. This type of demonstration is useful when an object's movement must be precise or when an object should change form over time (the SetImage function is used to show an object changing form).

Periodic transformation is specified by setting the "Every" box in the clock before a response demonstration. Such a demonstration was used to show the monster's movement in the Arcade game and the bottom car's movement in the simulator.

5.2.1 *A Notion of Direction.* An important contribution of *Pavlov*, illustrated by the example driving simulator, is that a designer can demonstrate animation in which the end-user not only initiates an object's movement, but accelerates it and changes its direction. Though such behavior is the primary activity in many game-like applications, there has been little research in this area, and commercial systems such as *Director* require extensive programming to develop this part of an application. *Pavlov* provides the capability to map any stimulus (movement of a slider, rotation of a gauge, change of a variable) to a response of accelerating or changing the direction of an object.

One important observation for the demonstration of animation is that many objects do not move in an arbitrary manner. Instead, they move forward and backward in the direction they are facing and rotate their base to turn. The standard Move(x,y) operation in a drawing editor is not sufficient for demonstrating the movement of such objects because it specifies both a distance and an absolute direction. To solve the problem, a notion of direction was added to the stimulus-response model. Designers can set a *"current direction"* attribute for an object, causing a direction arrow to be displayed on it during development. The object can then be moved in two ways: in real-time response mode, the object will follow the mouse by moving forward *and rotating*, and thus move naturally along a curved path (Figure 5). In all other modes, the object is restricted to moving along the vector defined by its direction attribute. This mechanism makes

Fig. 5. A directed object's nose follows the movement of the mouse.

it possible for the designer to demonstrate a *MoveForward(d)* operation, which is essential for demonstrating acceleration.

5.2.2 *Periodic Responses.* The notion of a periodic response is also necessary in demonstrating acceleration. In the driving simulator, when the end-user drags the accelerator, the car should begin moving and *continue* to *move*, even after the end-user releases the mouse. In *Pavlov*, the designer explicitly specifies continuous activity by setting the *Every:* box on the clock before a response operation.

Pavlov also provides special semantics so that a periodic MoveForward operation essentially causes the acceleration of an object. The following run-time rule is followed: the execution of successive periodic MoveForward operations on the same object results not in two alternating and possibly opposite actions, but in a single action combining the magnitudes of the operations. For example, in the driving simulator, when the car is already moving at 4 units/frame and the end-user manipulates the accelerator again, say, 25% of the way back toward the origin, it causes the response: periodic MoveForward (−1) units/frame. This second operation is combined with the existing one so that the car slows down to 4 − 1 = 3 units/frame, instead of alternating between moving forward 4 units, and backward 1 unit. If the end-user rotates the accelerator back to its origin (at the left edge of the enclosing rectangle), the periodic Move Forward obtains a parameter of 0, and the car stops.

The system only uses these semantics for periodic move-forward operations. For other operations, successive periodic responses will execute in tandem. For instance, by demonstrating two periodic move operations, the designer can demonstrate that an object can move back and forth, such as in an *animated move icon.*

5.2.3 *Direct Demonstration of Acceleration.* An alternative method of demonstrating acceleration has also been added to the *Pavlov* environment. After demonstrating a stimulus that should cause the acceleration, the designer enters *real-time* response mode and moves an object forward at the desired speed. Generally, a real-time response is used to demonstrate a fixed animation path as a response to a simple button-click or time. When a real-time MoveForward is demonstrated as a response to a transformational stimulus (one with parameters), the system does not record a series of discrete timestamped operations as usual, but instead records a single periodic operation. The distance moved each period is computed by dividing

the total distance of the demonstrated movement by the time of the demonstrated movement (d/t), and the period is set to 1 (ms).

The advantage of this scheme is that the designer truly demonstrates the speed of the movement; the disadvantage is it complicates the syntax of the system. In particular, with this alternative, an auxiliary key is necessary to distinguish whether a directed object should just move forward or should move forward and rotate as the mouse is dragged. A more thorough analysis of the advantages and disadvantages will be provided after more feedback is gathered from users.

5.3 Summary of Contributions to Animation

The mechanisms described in this section allow interactive animation to be demonstrated. *Pavlov*'s contributions are (1) the integration of timing controls into the stimulus-response environment, (2) the introduction of a notion of direction into a PBD system, and (3) the introduction of special semantics for periodic response demonstrations, so that acceleration can be demonstrated. *Pavlov* also contributes an internal stimulus-response model, described in Section 7, which integrates timing and animation with interaction.

6. THE PAVLOV EDITOR

Pavlov's editor provides a level of design with more expressive power than the demonstrational interface, but which is still at a high enough level so that most end-users can use it effectively. The editor is used to refine or correct behaviors inferred by *Pavlov*: the designer can enter complex (not just linear) formulas for response parameters and can enter formulas that refer to attributes of arbitrary objects in the interface. Using the editor is somewhat like coding, in that programming-language syntax is used to enter the response parameter formulas. However, the "coding" is in a strict context, much like a spreadsheet, and the designer is spared from the difficult part of animated interface programming, e.g., event loops, event dispatching, idle events, and timers interrupts.

The use of a second-level editor eliminates the need to constantly invoke dialogs to confirm inferences, as was done in early PBD systems such as Peridot [Myers 1988]. Other PBD systems provide editors based on property sheets [Halbert 1984], English-like stimulus-response descriptions [Myers et al. 1993], or storyboards without timing [Lieberman 1981].

Since *Pavlov* allows animation to be demonstrated, its editor borrows from traditional animation systems and is based on a timeline view of activity. However, because interaction is emphasized, *Pavlov* provides multiple timelines, each one showing the activities that occur in response to a particular stimulus. This method of organizing events by stimulus significantly eases the editing task. Without it, the designer would be required to specify the timed activity that is triggered by different stimuli in the same time-space, as is done in systems like *Director* (see Section 2).

The editor is shown in Figure 2 (Section 3.5). In order to view the operations that occur in response to a particular stimulus, the designer selects an object and a particular stimulus in the top-left list boxes. The objects that respond to that stimulus are then listed in the rows of the center matrix. The response operations executed for each object are then shown in the time-frames within the row. For example, in Figure 2, the matrix shows the operations executed in response to the up-arrow key being pressed while the main canvas is active. The response objects are the canvas and the object named "Bullet." An ellipse creation is shown as the response operation executed on the Canvas (shown as "0"). A series of timestamped move operations is shown as the response operations on the "Bullet."

The designer can select a particular response in a cell, and the inferred response parameters appear in the edit boxes labeled R1–R4. Any expression consisting of constants, stimuli parameters (s_i), and system-supplied object attribute functions (e.g., Object.Width) may be entered as a response parameter.

To view the operations that occur without an end-user stimulus (time-stimulated operations) the designer selects the "Time Stimulus" check box to the right of the stimulus list. Any objects that are created, transformed, or deleted without an end-user stimulus are then shown in the response matrix, along with the timestamped operations executed on them.

If a context has been demonstrated for when a stimulus should trigger a response, the developer can edit a textual representation of it by clicking the "Edit Condition" button. A dialog appears that allows the conditional functions inferred to be modified or deleted. For instance, if a developer had selected two conditions as the context for a response, *Pavlov* would, by default, AND the conditions together. Within the condition dialog, the designer can modify the condition by entering an equation involving any of the built-in conditional functions and relational operators.

The main contribution of the editor is that it is organized by stimulus, so that the various animation sequences are viewed and edited separately. Also, each cell represents an action, not an object state (as in *Director*), so animation paths are not coupled with a particular object or starting point. This scheme allows dynamic behaviors to be expressed in the editor, and it allows paths to be edited and applied to more than one object.

7. IMPLEMENTATION

This section describes *Pavlov*'s implementation, including the data structures and algorithms used to implement the stimulus-response model. The system was written in Visual C++ using the MFC library.

Pavlov's implementation is based on an augmented version of the traditional graphics object list used in most drawing editors. In the model, each canvas has an object list, and each canvas and object has not only an appearance component (e.g., location, color, etc.) but a behavior component in the form of a stimulus-response list (see Figure 6). Each stimulus-

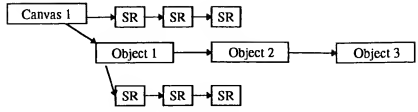

Fig. 6. Each canvas and object has a stimulus-response list.

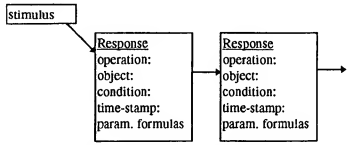

Fig. 7. A stimulus-response (SR).

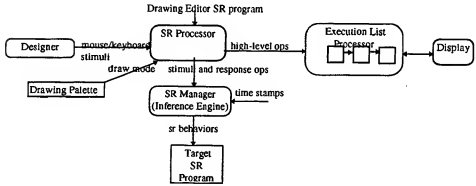

Fig. 8. Pavlov's development time architecture.

response (SR) contains a single stimulus and a list of 0 or more responses (Figure 7).

The responses record both the demonstrated operation and its parameters, the object of the operation, any conditions specified, a timestamp (which is relative to the time of the stimulus), and the parameter formulas that are inferred by the system from the demonstration and then used to compute the run-time parameters of the response.

7.1 Development Time Architecture

Pavlov's development time architecture (draw, stimulus, or either response mode) is shown in Figure 8. The drawing editor used to demonstrate

behaviors is itself modeled as an SR program. During development (all but Test mode) this program is fed into the *SR Processor* so that the designer can demonstrate graphical operations. This predefined program basically maps mouse drag events to the drawing editor operations (e.g., move), with a condition dependent on the drawing mode.

When in stimulus mode or either of the response modes, the *SR Processor* executes the high-level operations triggered by mouse or keyboard stimuli and sends records of these operations to the SR Manager, to be recorded and generalized. In stimulus mode, when an operation is concluded (e.g., a mouse-up event occurs) the system records the implicit stimulus-response mapping and the high-level stimulus:

```
Canvas::OnMouseUp(int mouseAndKeyFlags,CPoint curPoint)
{  if (m_DevelopmentMode == STIMULUS)
   {  if (m_ObjectWasDragged) // a transformation or a create operation
      {
   m_CurrentObject.SRManager.RecordImplicitSRMapping
   (mouseAndKeyFlags,m_CurrentOp)
   m_CurrentObject.SRManager.RecordStimulus(CurrentOp,
   curPoint.x-m_downPoint.x,curPoint.y-downPoint.y)
      // curPoint is the current mouse location. (m_downPoint is the
        location of mouse-down
      }
      else // a simple button click
        m_CurrentObject.SRManager.RecordStimulus
        (BUTTONUP,curPoint.X,curPoint.Y)
   }
}
```

For instance, if the designer selected move mode from the drawing palette, and used the left mouse button to move Object1, the first behavior in Figure 9 is recorded. In response mode, when an operation is concluded (e.g., a mouse-up event occurs), the function *RecordResponse* is called:

```
Canvas::OnMouseUp(int mouseAndKeyFlags,CPoint CurPoint)
{
   if (m_DevelopmentMode == RESPONSE)
   {
     if (m_ObjectWasDragged) // a transformation or a create operation
       m_CurrentStimulusObject.SRManager.RecordResponse
       (m_CurrentOp,m_CurrentObject,
       curPoint.x-m_downPoint.x, curPoint.y-m_downPoint.y)
   }
}
```

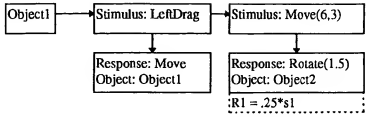

Fig. 9. A sample stimulus-response list.

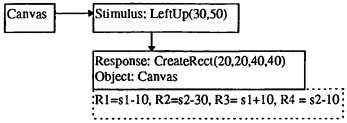

Fig. 10. A creation response.

RecordResponse both connects the response to the current stimulus and infers the run-time response parameter formulas.

A simple inferencing scheme is used, as introduced in Section 4. If both the stimulus and response operations are transformations, the response parameter formulas are built by computing proportional constants relating the demonstrated stimulus and response operations. For example, suppose that Object2 was rotated 1.5 radians as a response to the movement of Object1. Then in the call to *RecordResponse*, Object1 is the "CurrentStimulusObject"; the CurrentOp is a Rotate; and the CurrentObject is Object2. RecordResponse then modifies Object1's stimulus-response list as shown in Figure 10 (the inferred response parameter formula is in the dotted rectangle). In cases when the stimulus has two parameters and the response only one, the *RecordResponse* function checks if the first parameter of the stimulus is 0. If it is, the second parameter (e.g., the y-movement 3) is used in computing the proportional constant (e.g., response parameter formula "R1 = 0.5*s2" is computed).

If the stimulus is a mouse-up (button click) or a key press, constant values are used in the response formula instead of a proportion (e.g., the response formula would use the exact response demonstrated, e.g., "R1 = 1.5."

If the stimulus is a mouse up (button-click), and the response is a creation instead of a transformation, the system infers that the run-time response parameters are offset from the stimulus instead of proportional (Figure 10).

When a demonstrated response is a creation, the newly created object instance is added to the object list just as in draw mode. A *template* object is also created and is pointed to by the *instance*. When a stimulus is

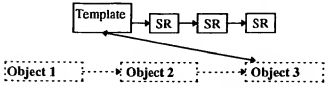

Fig. 11. Behaviors connected to a stimulus object.

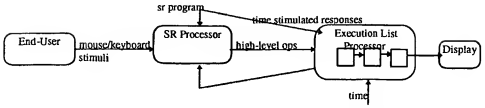

Fig. 12. The execution time (test mode) architecture of a developed SR program.

demonstrated on an instance, the resulting stimulus-response behaviors are connected not to the instance, but to the template object (Figure 11). When a response is demonstrated on an instance, the object of the response is recorded as the template. At run-time, the response is then executed on all instances of the template.

7.2 Run-Time Architecture

The execution time (test mode) architecture of a developed SR program is shown in Figure 12. The architecture illustrates two major modifications that were added to the original model (in the *DEMO* system) to handle timing and animation. The first is the addition of the execution list. In *DEMO*, when a stimulus occurred its responses were executed immediately. In an animated interface, however, responses are not necessarily executed immediately when their corresponding stimulus triggers them. Thus, in *Pavlov*, they are given a timestamp and are "held" in the execution list.

The second addition concerns the list of time-stimulated responses. Since these require no stimulus to occur, at the beginning of execution (test mode), they are placed in the *execution list* with their respective time-stamps.

Processing then proceeds in the following manner. For each mouse or keyboard stimulus that occurs, the *Sr processor* traverses the stimulus object's stimulus-response list to find the response operations associated with the stimulus. If the preresponse condition for this behavior evaluates to true (or there is no condition), these responses are copied into the *execution list* with a timestamp of $t + t_r$ where t is the time the stimulus occurred (the current time), and t_r is the recorded timestamp of the response. When a response is copied into the execution list, parameters for it are computed using the inferred response parameter formulas, the run-time stimulus parameters, and any object attributes referred to in the

Table I. Interface Behaviors and How Each is Specified in Selected Systems

Behavior	Pavlov	Marquise	Inference Bear	Director	KidSim
Constant Response	D	D	D	D+E+C	C
Proportional Response	D	D	D	C	C
Complex Response	D+E,C	D+E,C	D,C	C	C
Create/Delete Response	D	D	D	E+C	D
Conditional	D+E,C	D+E,C	D+C	C	D
Transform Over Time	D+E	C	C	D	D
Periodic Response	D+E	C	C	D+C	C
Interactive Animation	D	C	C	C	C
Acceleration Response	D	C	C	C	C

(D) means the behavior can be specified by demonstration; (E) means a second-level editor is used; (C) means coding is necessary; X+Y means a combination of approaches is necessary; X, Y means approach X can be used for most behaviors of the class, but that Y is required for some.

response parameter formulas. For instance, when the end-user moves Object1 with parameters (2, 3) in the example of Section 7.1, the run-time response is computed as a Rotate with parameter = 0.25*2 = 0.5.

When the system is not processing stimuli, i.e., there is an *idle event*, the *execution list* is processed. All operations whose timestamp is less than the current time are executed. After execution, a nonperiodic operation is removed immediately from the ready list; a periodic operation is left in the list, with its timestamp incremented by the size of its periodic interval. In either case, the executed operation is reentered into the *Sr processor* as a stimulus, so a chaining of events can occur.

The model does not guarantee that operations will be executed at the time specified, because the single processor is sometimes busy handling input. In practice, however, it provides *visually* acceptable performance on a Pentium processor.

8. A COMPARISON OF PBD SYSTEMS

One way to measure a PBD system is by describing the behaviors that can be demonstrated and, secondarily, the behaviors that can be specified with the system's second-level editor, if one exists. Any behaviors that cannot be specified in these interactive ways must be specified by a programmer modifying and extending the code generated by the PBD system.

Table I provides a list of interface behaviors and, for a selected set of development systems, stipulates which behaviors can be defined through demonstration (D), second-level editing (E), or coding (C).

The list does not focus on primitive operations (e.g., does a system allow sound or video?), but instead focuses on how the primitive behaviors of a system can be combined to create stimulus-response behaviors. The list contains behavior categories deemed important in (animated) interface design by this author. It is for comparison purposes only and should not be regarded as a definitive taxonomy. The set of development systems in-

cludes three stimulus-response systems (*Pavlov*, *Marquise*, and *Inference Bear*), an animation system (*Director*), and a graphical rewrite rule system (*KidSim*).

A basic interface behavior is modeled as follows:

```
O1.OnS(s1,s2. . . .)
{
  O2.R1(r1,r2, . . .)
  O3.R2(r1,r2. . .). . . .
}
```

O1, O2, and O3 are graphics objects. S is an end-user action or operation. R1 and R2 are operations, and OnS denotes the event-handling function that is called when event S occurs on object O1.

8.1 Constant, Proportional, and Complex Responses

The first three behaviors in the table concern the formulas of response parameters, i.e., the values of $r_1,r_2 \ldots$ A *constant response* is one in which the parameters r_i are constant values. A *proportional response* is one in which the parameters r_i are in proportion to the stimulus parameters s_i. A *complex response* is one in which the parameters r_i are not constant or proportional to the stimulus parameters, but are arbitrary functions of the stimulus parameters and the current attributes of any objects in the interface.

As shown in the table, all the stimulus-response systems listed (*Pavlov*, *Marquise*, and *Inference Bear*) allow constant and proportional behaviors to be demonstrated. Both *Pavlov* and *Marquise* require a combination of demonstration and second-level editing (D+E) to specify most complex behaviors; coding is necessary for others that concern mathematical functions or nongraphical data (e.g., from a database). *Inference Bear* can infer some complex behaviors because it allows multiple examples to be demonstrated. None of the systems allow all complex behavior to be specified without coding.

Director requires editing with the timeline and some scripting to specify a constant response behavior: the designer must enter a goto statement in the event-handling function to show what frame should be jumped to in response to the event. For proportional and complex behaviors, more extensive scripting is necessary.

Because *KidSim* does not focus on end-user interaction, all constant, proportional, and complex behaviors must be coded.

8.2 Creation/Deletion Response

A creation/deletion behavior is one in which the response operation R causes an object to appear in or disappear from the interface. All of the systems shown in the table, other than *Director*, allow creation and deletion operations to be demonstrated. By editing the timeline, a *Director* designer can specify when a single object appears and disappears, but coding is necessary to specify that instances of a class of objects can be generated.

8.3 Conditional Response

A conditional response behavior is of the form

```
O1.OnS(s1,s2. . ..)
{
  if (C)
    O2. R(r1,r2. . .)
}
```

where C is some predicate.

In *Inference Bear*, conditions are added by hand (coded) to stimulus-response code. *Marquise* identifies some conditions automatically by allowing responses to be connected to markers tracking the mouse during a stimulus demonstration. In *Pavlov*, the designer demonstrates conditions during a stimulus-response sequence, then specifies which of the identified conditions should modify the stimulus-response with a dialog. With the dialog, conditions can be specified as any relation consisting of binary relationships between objects, integer variable relations (e.g., mode = 2), and AND, OR, and NOT operators. Other conditions must be coded. *KidSim* allows conditions to be specified through the demonstration of graphical rewrite rules (see Section 4.2.3). *Director* requires scripting to specify all conditional behaviors.

8.4 Timing and Animation

The last four behaviors listed in Table I correspond to timing and animation behaviors. As the table illustrates, *Pavlov* is the only stimulus-response system to focus on the interactive specification of timing and animation; in *Marquise* and *Inference Bear*, such behaviors must be coded.

A *transform-over-time* behavior is one in which objects transform in the interface at some rate. *Pavlov* and *Director* allow fine-grained control over such behaviors: through demonstration and editing, a behavior in which responses execute at some time t after a stimulus can be specified:

```
O1.OnS(s1,s2. . ..)
{
  R1(r1,r2. . .)AT t1
}
```

In *KidSim*-developed simulations, objects move every time-frame if a graphical rewrite rule fires; there is not a mechanism for specifying fine-grained control of the timing of an animation (e.g., that an object moves every 10 frames, no matter what).

A *periodic response* behavior is of the form

```
On O1.S
{
   O2. R(r1,r2,. . .) every P until L
}
```

It models behaviors in which a stimulus triggers an object to begin *and continue* to transform over time.

Of the systems shown in Table I, only *Pavlov* allows periodic behaviors to be specified without coding. There are two methods used: a form-based one involving the editor and one based on demonstration. In the first method, the period "P" is set in the development clock during a demonstration or in the editor after a demonstration. The limit "L" must be set with the editor. It cannot be specified that the period "P" or limit "L" change dynamically, and the parameters of the response cannot refer to the current period. Thus, for instance, one could not interactively specify a loop that creates a series of lines in which each line is located some distance, proportional to the line number, from the initial one. The demonstrational method generates a periodic response from a real-time response demonstration (see Section 5.2.3). The period P is automatically set to 1, though it, the limit "L," and the response parameters can be edited after the demonstration.

An *interactive-animation* behavior is one in which the end-user triggers and/or controls an animation sequence. *Pavlov* allows such behavior to be demonstrated. *Director* requires goto statement scripting to specify animation triggered by button clicks. More complex triggers or animation control requires more extensive scripting. *KidSim*, *Marquise*, and *Inference Bear* do not allow interactive animation to be demonstrated.

An *acceleration response* behavior is one in which the end-user controls the speed of an animation. *Pavlov* allows such behaviors to be demonstrated because of the notion of direction and because of the way periodic response demonstrations are interpreted (see Section 5). None of the other systems allows acceleration to be defined without coding.

9. USER STUDY

An informal user study was performed to compare the efficiency of developing animated interfaces with *Pavlov* as opposed to a standard interface development environment, in this case *Visual C++*. As opposed to our previous user study [Wolber and Fisher 1991] that focused on nonprogrammers, the subjects for this study were Master's-level computer science students. Each of the students had programmed with C++ for at least two years, with *Visual C++* for a year, and each had implemented a drawing editor using the Visual C++ environment. The students were familiar with *Pavlov*, but had never used it.

9.1 Task Description

The task was to create the Arcade game, first with *Pavlov* and then with Visual C++. The arcade game was chosen because it exhibits many behaviors found in animated interfaces: the objects in the interface are application specific and not standard widgets; the objects move around the screen at different speeds; and action is triggered by both end-user events and context. Positive results were expected because the development of such

interfaces is the strength of *Pavlov*, whereas Visual C++ is stronger for developing interfaces with standard widgets and links to computational code.

The students were given a *Pavlov* user's manual and were allowed to practice with the system for one hour prior to the test and before being given the task. The manual contained only basic descriptions of *Pavlov*'s features, and no examples concerning how to build a complex interface (such as the Arcade game) were provided. After this session, students were given a textual description of the task (similar to that in Section 3) and were shown execution of an implementation of the Arcade game. During the test they were allowed to ask technical questions about *Pavlov* or Visual C++. Specific questions concerning how to complete the task were not allowed.

9.2 Results

Four students took part in the study. Two pieces of data were recorded: the time to complete each project and each student's estimate of how long it would take to complete a project of similar complexity the following day (after becoming "experts" in *Pavlov* and in *Visual C++* graphics/timing programming). The average time to complete the project with *Visual C++* was 330 minutes. The average time to complete it with *Pavlov* was 80 minutes. The students estimated, on average, that completing a similar project after the experience would require 150 minutes with *Visual C++* and 17.5 minutes with *Pavlov* (two expert *Pavlov* designers also implemented the Arcade game; both required less than 10 minutes).

Though more formal user studies are necessary, the experience suggests that stimulus-response techniques can have a dramatic impact on productivity. These results are of course tempered by the fact that the test was performed for only a single application. After code generation facilities are added to *Pavlov*, a more thorough test will be conducted. The test will involve a suite of interfaces and will include *Director* as well as *Visual C++* (and any future PBD systems that incorporate both animation and interaction capabilities). The test suite will include not only small, very graphical interfaces such as the Arcade game and driving simulator, but also larger-scale applications that include a significant computational element.

The experience also suggests that significant training time (at least an hour or two) is necessary to become a competent *Pavlov* designer. Though the basic stimulus-response concept was quickly understood by the test subjects, early in the session they had difficulty manipulating the different mode palettes and understanding the more complex concepts such as conditions. To complete the Arcade example, they also had to learn how to use the second-level editor. Much of the 80 minutes (on average) and the preliminary hour was spent becoming familiar with the environment. By the end of the session, however, it appeared that the subjects were accustomed to both the demonstrational and editing interfaces (this is vaguely supported by their average estimate of 17.5 minutes to complete an Arcade game-like interface following their experience).

10. FUTURE DIRECTIONS

Pavlov can be used to quickly create some animated interfaces and prototype others. But in order for a PBD system to be used for large-scale development, it must generate code and include code-editing facilities similar to that provided in Visual C++ and other standard interface builders. The capability to generate multithreaded Java code from a *Pavlov*-development session is currently being implemented. Though the work is relatively straightforward from a research standpoint, the resulting system can be used as part of user tests to address one of the biggest criticisms of PBD: that they can only be used to create "toy" applications.

A limitation of *Pavlov* that we feel can be addressed at the demonstrational level is that it focuses on the objects that are transformed and does not provide support for designing the paths on which objects move. Currently, there is no method of defining an arbitrary path object, as in Repenning [1995]. If such a facility were added to *Pavlov*, a designer could draw a path and then demonstrate that some stimulus should cause an object to *move forward along the path*. Such a facility would allow demonstration of, for example, a Ferris wheel and the restricted movement in a board game.

More work is also necessary in evaluating and integrating the existing PBD techniques introduced in *Pavlov* and other systems. Planned studies include (1) a thorough analysis of the difference between single- and multiple-example systems, including comparisons of expressive power and usability, and (2) a detailed examination of the difference between *Pavlov*'s method of demonstrating context and the method used in graphical rewrite rule systems.

11. SUMMARY

The main contribution of *Pavlov* is that it allows end-user interaction, context, and timing to be specified within a single cohesive framework, stimulus-response demonstration. It also introduces a stimulus-response version of the traditional key-frame timeline, thus integrating PBD stimulus-response technology with that of conventional animation systems.

The main difference between *Pavlov* and other PBD systems is the inclusion of timing and thus the ability to demonstrate animated interfaces. Various techniques are introduced, including those in which the end-user controls the direction and speed of object movement.

The main difference between *Pavlov* and graphical rewrite systems is the inclusion of end-user interaction and timing. With the stimulus-response model, context is only demonstrated as a condition on when a stimulus triggers a response. The designer can also demonstrate behaviors in which action is triggered by end-user activity or time. Thus interactive animation, such as that found in games, can be designed without coding.

There are two main differences between *Pavlov* and animation systems such as *Director*. The first is that interaction is specified by demonstration instead of coding with a scripting language. The second is that the

timeline-based editor in *Pavlov* organizes the animated activity by the stimulus that triggers it, instead of requiring the designer to organize it manually within a single timeline.

REFERENCES

BAECKER, R. 1969. Picture-driven animation. In *Proceedings of the Spring Joint Computer Conference*. AFIPS Press, 273–288.

CYPHER, A., Ed. 1993. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, Mass.

CYPHER, A. 1991. Eager: Programming repetitive tasks by example. In *Proceedings of CHI '91*. ACM, New York, 33–39.

FISHER, G., BUSSE, D., AND WOLBER, D. 1992. Adding rule based reasoning to a demonstrational interface builder. In *Proceedings of UIST'92*. 89–97.

FISHER, G., SUNG, H., NGUYEN, S., AND NGUYEN, T. 1996. Animation in a demonstrational interface builder. In *Proceedings of CHI '96*. ACM, New York, 267–268.

FRANK, M., PIYAWADEE, S., AND FOLEY, J. 1995. Inference Bear: Designing interactive interfaces through before and after snapshots. In *Proceedings of DIS '95*. 167–175.

GOULD, L. AND FINZER, W. 1984. Programming by Rehearsal. *Byte 9*, 6.

HALBERT, D. 1984. Programming by Example. Ph.D. thesis, Dept. of Electrical Engineering and Computer Science, Univ. of California, Berkeley, Calif.

KURLANDER, D. AND FEINER, S. 1993. Inferring constraints from multiple snapshots. *ACM Trans. Graph. 12*, 4 (Oct.), 277–304.

LIEBERMAN, H. 1981. Tinker: Example-based programming for artificial intelligence. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI)*.

LIEBERMAN, H. 1985. Video games by example. *SIGGRAPH '85 Video Rev. 12*, 1.

MAULSBY, D., WITTEN, I., KITTLITZ, K., AND FRANCESCHIN, V. 1992. Inferring graphical procedures: The Compleat Metamouse. *Hum. Comput. Interact. 7*, 1, 47–89.

MYERS, B. 1988. *Creating User Interfaces by Demonstration*. Academic Press, San Diego, Calif.

MYERS, B., MCDANIEL, R., AND KOSBIE, D. 1993. Marquise: Creating complete user interfaces by demonstration. In *Proceedings of INTERCHI '93*. ACM, New York, 293–300.

OLSEN, D., AHLSTROM, B., AND KOHLERT, D. 1995. Building geometry-based widgets by example. In *Proceedings of CHI '95*. ACM, New York, 35–42.

REPENNING, A. 1995. Agent sheets: A medium for creating domain-oriented visual languages. *Computer 28*, 17–25.

SMITH, D. C. 1977. *Pygmalion: A Computer Program to Model and Stimulate Creative Thought*. Birkhauser, Basel.

SMITH, D. C. AND CYPHER, A. 1995. KidSim: End-user programming of simulations. In *Proceedings of CHI '95*. ACM, New York, 234.

STASKO, J. 1991. Using direct manipulation to build algorithm animations by demonstration. In *Proceedings of CHI '91*. ACM, New York, 307–314.

WOLBER, D. 1996. Pavlov: Programming by stimulus-response demonstration. In *Proceedings of CHI '96*. ACM, New York, 252–259.

WOLBER, D. AND FISHER, G. 1991. A demonstrational technique for developing interfaces with dynamically created objects. In *Proceedings of UIST '91*. 221–230.

Received July 1996; revised August 1997; accepted August 1997